アジリティのある探索的形式仕様記述のための テストフレームワーク

小田 朋宏
株式会社 SRA
tomohiro@sra.co.jp

荒木 啓二郎
熊本高等専門学校
araki@kyudai.jp

要旨

VDM-SL は実行可能なサブセットを持つ形式仕様記述言語である．我々は，形式仕様記述工程の初期段階における探索的な記述プロセスに注目し，それを支援するための仕様記述環境として *ViennaTalk* を開発してきた．探索的な仕様記述では仕様には頻繁に誤りが発生するため，仕様記述の生産性と品質向上のためには，誤りの早期発見と効率的な除去が求められる．本稿では，探索的な試行錯誤の中で *VDM-SL* 仕様の記述誤りを早期に発見するためのテストフレームワーク *ViennaUnit* について，*ViennaTalk* での実装を説明し，ツール機能としての応用と評価を示す．

1. はじめに

ソフトウェアシステムの仕様を形式的に記述することで 仕様記述の問題点をより早期に発見し修正する開発手法として, 形式手法が注目されている [1, 2]．VDM-SL [3] は実行可能なサブセットを持ち，複数のインタプリタ実装および自動コード生成器が提供されている形式的仕様記述言語である [4]．実行可能な VDM 仕様は vdmUnit 等のテストフレームワークを利用して単体テストを継続的に行うことで，高い品質の仕様記述が可能であることが事例から知られている [5].

筆者らはこれまで仕様記述工程の初期段階に着目し，探索的に形式仕様記述を遂行することを支援する環境 ViennaTalk[6] を開発してきた．探索的な仕様記述においては試行錯誤が多く行われ，問題に対する新たな知見に基づく仕様記述の変更が頻繁に行われる．そのため，高いアジリティが求められる．ViennaTalk は，Smalltalk 処理系の 1 つである Pharo[7] 環境上に構築されたメタ IDE であり，VDM-SL の外部インタプリタを利用するためのブリッジ，VDM-SL 仕様から Smalltalk ライブラリソースを出力する自動コード生成器，パーサ，構文ハイライトや自動整形機能付きのエディタ部品など，VDM-SL 向けの様々なツールを開発するためのコアとなるコンポーネントを提供している．現在 ViennaTalk 上で利用可能なツールとして，WebIDE サーバ VDMPad，VDM-SL 仕様の編集およびライブ実行をおこなう VDMBrowser，仕様の妥当性を確認するための UI プロトタイピング環境 Lively Walk-Through, Web API プロトタイプ環境 Webly Walk-Through がある．本稿では，ViennaTalk が提供するツールのうち VDMBrowser を取り上げて，探索的仕様記述に適したテストフレームワーク ViennaUnit と VDMBrowser での実現を説明する．そして，VDMBrowser での実現の評価として，仕様の誤りの早期発見につながるかを分析し議論する．

2. VDM-SL および ViennaTalk の概要

VDM-SL は ISO により標準化されたモデル規範型の形式的仕様記述言語で，記述対象を抽象するための型，定数，関数を定義し，状態空間の定義と状態を変更する操作を定義することで，システムの入出力や状態変化を記述する．VDM-SL にはモジュール機能があり，各モジュールは imports 宣言 および exports 宣言により外部インターフェイスを宣言する．VDM-SL で提供されている型には，自然数型，非 0 自然数型，整数型，小数型，文字型，ブール型，引用型，トークン型などの基本型と，複合型，列型，集合型，直積型，合併型，オプション型，

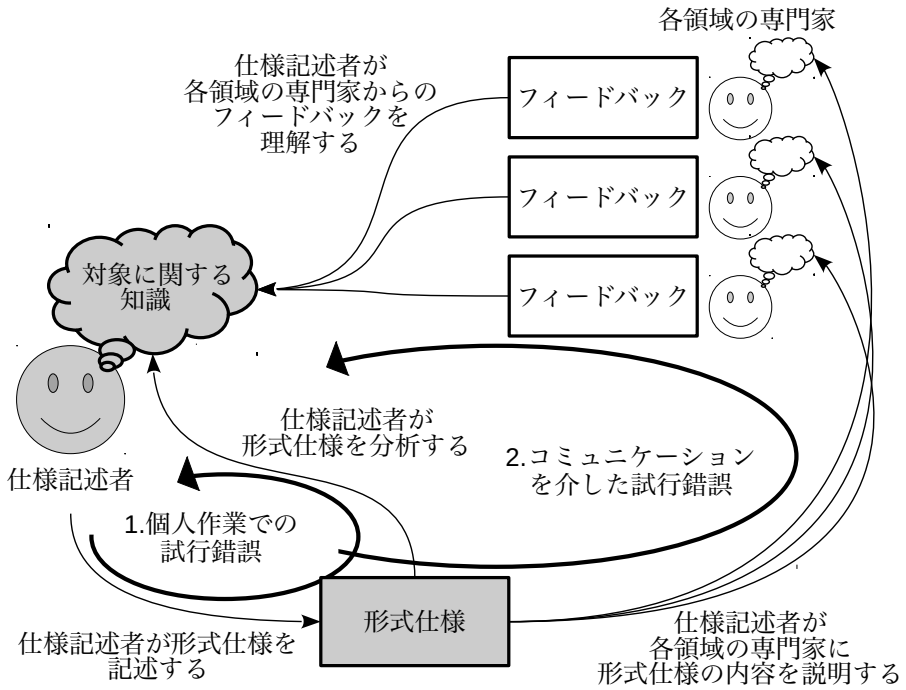

図 1. 探索的仕様記述における試行錯誤のプロセス

関数型などの合成型がある．文字列のための型はなく，文字列は文字型の列 `seq of char` として扱う．VDM-SL での関数とは参照透明な関数であり，操作とは状態への参照や破壊的代入を伴う手続きである．VDM-SL ではそれらの関数や操作に対して事前条件および事後条件を表明することで，それに対応する実装が満たすべき性質を定義する．また，型および状態に対して不変条件を表明することができる．VDM-SL は実行可能なサブセットを持つことから，プログラミング言語と同様にインタプリタ実行もしくはコード自動生成器を通してプログラムとして実行することも可能であるが，不変条件，事前条件および事後条件からなる表明が形式仕様としての重要な役割を果たす．

ViennaTalk は，Smalltalk 処理系の 1 つである Pharo[7] 環境上に構築された探索的仕様記述 [6] 環境である．探索的仕様記述とは形式仕様記述工程の初期段階に見られる試行錯誤を伴う工程であり，形式仕様が顧客の要求および対象ドメインに適合し，機能項目の記述漏れがないことを目標とする．探索的仕様記述に続いて精緻化により，機能定義について未定義部分や矛盾がない厳密な仕様を記述する．

図 1 に探索的仕様記述のプロセスを示す．探索的仕様記述では，個人作業における試行錯誤と，コミュニケーションを介した試行錯誤の，2 種類の試行錯誤が繰り返し行われる [6]．個人作業における試行錯誤は，仕様記述者が持つ記述対象に関する知識に基づいて，どのような機能項目をどのように定義するかについて，適切な記述を模索しながら進める作業である．コミュニケーションを介した試行錯誤は，作業中の仕様を各領域の専門家に説明し，フィードバックを得ることによって，より適切な記述を模索する作業である．これら 2 つの試行錯誤を通して，仕様記述者が対象に関する知識を深めていくことで，顧客の要求および対象ドメインに適合した仕様を作成する．ViennaTalk は探索的仕様記述に特化した仕様記述環境であり，形式仕様の初期段階を形成するための試行錯誤を支援する記述環境や，顧客および対象ドメインの専門家などからのフィードバックを得るためのプロトタイピング環境をツールとして提供している．

本節では特に，本稿のテーマであるテストフレームワークを利用するためのツールである VDMBrowser を説明する．VDMBrowser は探索的仕様記述を支援する仕様記述環境であり，大きな特徴として，仕様を記述す

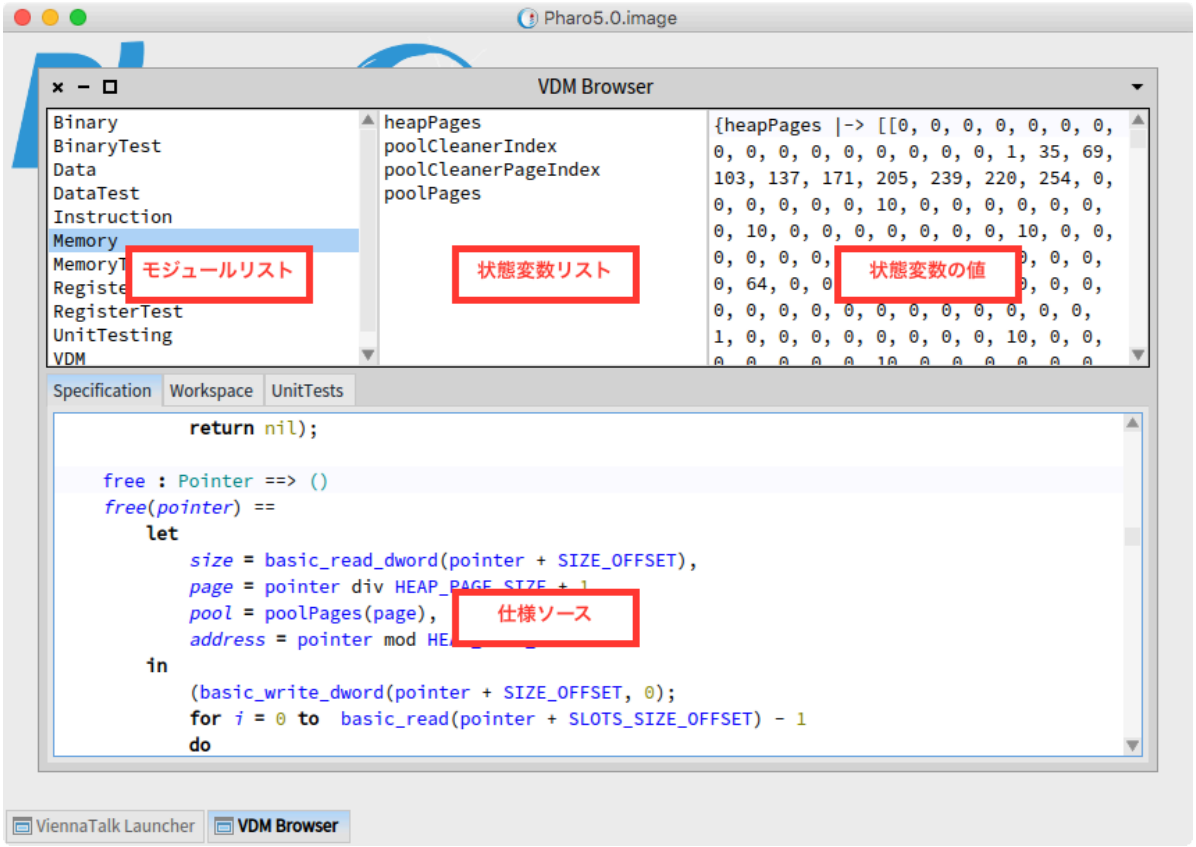

図 2. ViennaTalk 環境の画面例

るだけでなく，記述中の仕様のアニメーションが常に実行中であることが挙げられる．アニメーションとは，実行可能な仕様をインタプリタもしくは自動コード生成によって，当該仕様を実装したシステムの動作の模擬実行を指す．すなわち，一般的な開発環境でのテキストエディタの多くがテキストファイルを編集操作の対象としているのに対して，VDMBrowser が編集操作をする対象は VDM-SL 仕様を実装したものとして模擬実行している仮説的なシステムであり，仕様の編集とは模擬実行中のシステムのモデルを変更することを意味する．仕様記述者は VDMBrowser 上で模擬実行中のシステムの状態を問い合わせたり，状態を編集したり，特定の操作を模擬実行することができる．そして，システムの動作の具体例を見ながら，新たな機能項目を定義したり，既存の機能の定義を変更することができる．

図 2 に VDMBrowser の画面例を示す．VDMBrowser の画面は大きく上段と下段に分けられる．上段には左からモジュールリスト，状態変数リストおよび状態変数の値が表示される．VDM-SL の仕様はモジュール化されており，模擬実行中のシステムのうちどのモジュールに対してモデリングするかをモジュールリストによって選択する．上段中央の状態変数リストでは，システムの状態を表現した変数の一覧が表示され，ここで選択された変数の値が上段右側のエディタ上に表示される．上段右側の状態変数の値を編集し保存することで，模擬実行中のシステムの状態を変更することができる．これは VDM の他の統合開発環境である VDMTools[8] や Overture tool[9] にはない機能である．

下段には Specification，Workspace および UnitTests の 3 つのタブがあり，切り替えて使用する．Specification はモジュールリストで選択されたモジュールのソースを編集するエディタで，構文ハイライト，ソースの自動整形，表現式の評価実行などを行うことができる．Workspace はモジュールとは独立したテキストエディタで，探索的テストを行なったり，システムの利用シナリオを記述するために利用する．Workspace 上には任意のテキストを入力することができ，その一部を選択して VDM-SL 表現式として評価実行することができる．UnitTests は ViennaUnit によるテストの結果を一覧表示する．ViennaUnit については，第 3 節で説明する．

3. ViennaUnit

仕様の誤りは 2 つの試行錯誤のいずれにおいても発生し，いずれにおいても発見され得る．コミュニケーションを介した試行錯誤では，仕様の誤りがあまりにも多いと，他の開発関係者からの有効なフィードバックを得ることが困難になる．したがって，個人作業での試行錯誤の中で，仕様中の誤りを発見し修正することが求められる．

プログラミングにおいては，xUnit[10] をはじめとする自動テストによる誤りの発見が有効であり，実践されている．xUnit によるテスト駆動開発では，テストプログラムを一種の仕様と見做して，テストに合格するプログラムを記述することを直近の目標とする．ただし，xUnit でのテストプログラムはプログラム実行の例示であり，VDM-SL による系統的に抽象された仕様とは異なる．実行可能なサブセットを持つ VDM-SL でも，xUnit と同種の自動テストのためのツールが複数提供されている．VDM-SL 仕様にはプログラムの機能に関する表明が系統的に記述されることから，プログラミング言語での xUnit 用に記述するテストコードに記述される表明の多くは，既に仕様中に記述されていることが多い．VDM-SL 仕様がプログラムが持つべき性質を抽象し系統的に記述したものであるのに対して，VDM-SL 仕様に対するテストはそれらの性質に対する具体例を示して一貫性を確認する作業である．テストで確認すべき具体例は，仕様を模索する探索的仕様記述の段階では顧客やドメイン専門家の理解に基づいた特徴を表すことが求められ，高品質な仕様を定義する精緻化では確認の漏れがないよう機能定義に対して網羅的であることが求められる．

xUnit と同種のテストフレームワークに vdmUnit がある．vdmUnit は実際には，2 つのテストフレームワークが存在する．1 つは VDMTools[8] 上で動作するテストフレームワーク [11] であるが，VDM のオブジェクト指向拡張である VDM++が対象であり，VDM-SL はサポートされていない．もう 1 つは Overture tool[9] に付属するライブラリ [12] で，Java コード自動生成器と連携して JUnit 用のソースコードを出力する．Java コードの自動生成と Java プログラムのビルドおよび JUnit 実行というステップを踏まなければならず，頻繁な自動テスト実行による早期の誤り発見には向かない．

VDM で利用可能なテスト技法に，組み合わせテストがある．大量の組み合わせを効率よく実行するため，仕

様の各機能項目について，未定義な組み合わせがないか，条件に漏れがないかを確認することができる．しかし，VDM での組み合わせテストの目的は機能項目での未定義な組み合わせや条件漏れの発見が有効であるのは探索的仕様記述の後工程である精緻化であり，探索的仕様記述の段階ではまだ機能項目の定義における網羅性は必須ではない．また，組み合わせテストでは大量の組み合わせについてアニメーション実行を行うため，大きな計算リソースが求められる．

本稿で提案する ViennaUnit は，VDM-SL 向けの xUnit の簡略な実装である．ViennaUnit は，探索的仕様記述での利用を想定し，頻繁な仕様記述の変更に対して高いアジリティで追従して早期の誤り発見を可能にすることを目的としている．通常のプログラミング言語での xUnit では多くの表明が記述される．VDM-SL では仕様に不変条件，事前条件および事後条件などの多くの表明を持つ．ViennaUnit で記述する表明は主に仕様が要求されたシナリオを遂行可能であるかの確認であり，シナリオで約束した特性を満たすことの確認事項である．ViennaUnit で記述される表明の多くは仕様中に記述された表明と重複して，シナリオの粒度での意図の記述であることから，ViennaUnit での自動テストは表明のダブルチェックと見なすことができる．

ViennaUnit の実行メカニズムは非常に単純に作られている．ViennaUnit は構文的には通常の VDM-SL 仕様と同じ文法によって記述される．仕様が定義するモジュールのうち，モジュール名が `Test` で終わるものをテストモジュールとみなし，テストモジュールが定義する操作のうち，操作名が `test` で始まるものをテスト操作とする．

ViennaUnit を実行すると，対象となる仕様から全てのテスト操作を列挙し，それぞれアニメーション実行する．テスト操作を実行中に `AssertFailure` 型 または `AssertEqualsFailure` 型の例外が発生したらテスト結果は「失敗」となる，テスト操作を実行中に VDM-SL の処理系によって実行時エラーが発生したらテスト結果は「エラー」となる．テスト操作が正常終了した場合には「成功」と判定される．

ViennaTalk では，テスト操作を記述するための補助となるモジュールとして，`UnitTesting` モジュールを提供している．図 3 に `UnitTesting` モジュールの VDM-SL ソースを示す．`UnitTesting` モジュールは `assert` および `assertEquals` の 2 つの操作を提

```
module UnitTesting
exports all
definitions
types
     AssertFailure :: msg : seq of char;
     AssertEqualsFailure ::
         actual : ?
         expected : ?
         msg : seq of char;

operations
    assert : bool * seq of char ==> ()
    assert(b, msg) ==
        if not b
        then exit mk_AssertFailure(msg);

    assertEquals : ? * ? * seq of char ==> ()
    assertEquals(actual, expected, msg) ==
        if
            actual <> expected
        then
            exit mk_AssertEqualsFailure(
                actual,
                expected,
                msg);

end UnitTesting
```

図 3. UnitTesting モジュールのソース

供する．`assert` 操作は引数として真偽値およびメッセージを取り，渡された真偽値が `false` であった場合に `AssertFailure` 型の例外を発生させる．メッセージは `AssertFailure` 型の値の中に格納される．`assertEquals` 操作は引数として 2 つの値とメッセージを取る．2 つの値は任意の型で良く，等しくない場合には `AssertEqualsFailure` 型の例外を発生させる．2 つの値およびメッセージが `AssertEqualsFailure` 型の値の中に格納される．`assertEquals(x, y, msg)` は `assert(x = y, msg` と記述することも可能だが，`assertEquals` 操作を用いることで失敗時に `x` と `y` の具体的な値を例外に格納してデバッグのための情報として利用することができる．

図 4 にテストモジュールの例を示す．仮想機械の開発のために記述された VDM-SL 仕様から抜粋した．図 4 の `MemoryTest` モジュールは，`UnitTesting` モジュールを使って，`Memory` モジュールに対する 2 つのテスト操作を定義している．`Memory` モジュールは仮想機械のメモリモデルを定義するモジュールであり，メモリアライメントやデータの読み書きを規定する．`test_align` 操作では，メモリアライメントに対するテストとして，サ

```
module MemoryTest
imports
    from Memory all,
    from UnitTesting
        operations
            assertEquals renamed assertEquals;
            assert renamed assert;
exports all
definitions
operations
    test_align : () ==> ()
    test_align() ==
        (for i = 1 to  Memory`ALIGNMENT
        do assertEquals(
            Memory`align(i),
            Memory`ALIGNMENT,
            "0");
        for i = Memory`ALIGNMENT + 1
        to  Memory`ALIGNMENT  * 2
        do assertEquals(
            Memory`align(i),
            Memory`ALIGNMENT  * 2,
            "1"));

    test_basic_read : () ==> ()
    test_basic_read() ==
        (Memory`addPage();
        Memory`basic_write(
            0x12,
            0xfedcba9876543210);
        assertEquals(
            Memory`basic_read(0x12),
            0xfedcba9876543210,
            "basic_read 64bits immediate");
        Memory`basic_write_byte(0x10, 0x01);
        Memory`basic_write_byte(0x11, 0x23);
        Memory`basic_write_byte(0x12, 0x45);
        Memory`basic_write_byte(0x13, 0x67);
        Memory`basic_write_byte(0x14, 0x89);
        Memory`basic_write_byte(0x15, 0xab);
        Memory`basic_write_byte(0x16, 0xcd);
        Memory`basic_write_byte(0x17, 0xef);
        assertEquals(
            Memory`basic_read(0x10),
            0xefcdab8967452301,
            "basic_read little endian"));

end MemoryTest
```

図 4. ViennaUnit で記述したテストモジュールの例

イズ 1 からアライメントサイズまでは，アライメントサイズまで拡張され，アライメントサイズより大きくアライメントサイズの 2 倍未満まではアライメントサイズの 2 倍に調整されることを確認する．test_basic_read 操作では，メモリページ上への 64 ビット整数の読み書きの結果が整合することを確認し，バイト単位で書き込んで 64 ビット整数として読むとリトルエンディアンとして解釈されることを確認する．

VDM-SL 仕様に対するユニットテストでは，test_align 操作のようにどのような結果をもたらす定義がされるべきか，あるいは，test_basic_read 操作のようにどのようなシナリオを受け付けてどのような結果をもたらす操作群が定義されるべきかをテストコードという形で規定する．例えば，test_basic_read 操作は Memory モジュールの basic_read 操作がメモリページ中の連続した 8 バイトをリトルエンディアンで 64 ビット符号なし整数として解釈することを確認する．Memory モジュールの basic_test 操作の定義においてもリトルエンディアンであることは規定されているが，アセンブリ言語プログラマの要請として，1 バイト単位で書き込む basic_write_byte 操作との組み合わせによってエンディアンを明確に示し，basic_read 操作の定義がシナリオの中でそれに適合していることを確認することが test_basic_read 操作の目的である．

VDM-SL 仕様に対するユニットテストは，仕様記述者やドメイン専門家の理解と機能モデルの間の整合性を確認する作業であると言える．すなわち，ユニットテストの失敗を早期に検出することが，仕様記述者の理解と仕様記述内容の乖離を早期に発見し，正しい理解および正しい機能定義に導くことにつながる．

4. VDMBrowser における ViennaUnit の実行とフィードバック

ViennaTalk の VDMBrowser は探索的仕様記述での試行錯誤を伴う仕様記述を支援する．したがって，VDM-Browser が扱う VDM-SL 仕様には誤りが頻繁に混入する．これは試行錯誤による編集の頻度の高さに加え，仕様自体がまだ未成熟であること，仕様記述者の対象への理解が正確ではないことに起因する．仕様の誤りを早期に発見して修正するためには，高い頻度でのユニットテストの実行が有効であると考えられる．

VDMBrowser では，仕様の修正があるたびに自動的に

バックグラウンドで ViennaUnit を起動し，全てのテストモジュールの全てのテスト操作を実行する．ただし，自動実行するかどうか，および，全てのテストモジュールを対象とするかまたは修正に直接関係するテストモジュールのみ実行するかは，設定により選択することができる．また，モジュールリストのメニューからユーザーの指示によって，全てのテストモジュールまたはモジュールリストで選択されたモジュールに対応するテストモジュールを実行する．本稿では，全てのテストモジュールを自動実行した場合について議論する．

ViennaUnit の実行はバックグラウンドで行われる．また，テスト実行の状態空間は，VDMBrowser 上のアニメーションの状態空間とは独立している．テスト実行中も VDMBrowser 上で仕様を編集したり，あるいは，VDMBrowser 上のアニメーションの実行を行うことができる．

ViennaUnit によるテスト結果は VDMBrowser 上の UnitTests タブに切り替えることで確認することができる．図 5 に ViennaUnit の結果の画面例を示す．リストの各行に，それぞれ 1 つのテスト操作の結果が表示される．成功したテスト操作は，緑色の丸アイコンに続いてテストモジュール名，テスト操作名および OK というメッセージで示される．表明に失敗したテスト操作は，黄色の丸アイコンに続いてテストモジュール名，テスト操作名およびエラーメッセージで示される．`assertEquals` に失敗した場合には，具体的に一致しなかった値のペアが示される．エラー終了したテスト操作は，赤色の丸アイコンに続いてテストモジュール名，テスト操作名およびエラーメッセージで示される．不変条件，事前条件，事後条件などのテスト対象の仕様中での表明失敗もエラーとして扱われる．

ViennaUnit は，失敗またはエラーとなるテスト操作を発見すると，ViennaTalk 環境の周辺領域に，仕様記述タスクの邪魔をせずにテストの失敗を知らせる通知を表示する．図 6 に通知の例を示す．この通知はウィンドウシステムのフォーカスを奪わないため，ユーザは VDMBrowser などでの編集作業を継続することができる．また，通知は半透明で表示され，数秒で透明度が高くなり消滅することで，ユーザへの干渉を小さく抑えている．

5. 評価

VDM-SL を用いた仕様記述を行っている開発作業において，ユニットテストの結果をログに記録するよう改造した VDMBrowser を 4 時間使用して，ViennaUnit の自動実行の有効性を評価した．改造した VDMBrowser は，仕様が保存されるたびに，自動的にバックグラウンドで ViennaUnit を実行し，結果をユーザには表示せずにログに記録した．ユーザは，ユニットテストの実行が必要だと判断した時に明示的に ViennaUnit を実行し，明示的な ViennaUnit の実行であることをログに記録した上で，そのテスト結果をログに記録した．

表 1 に，自動的なテスト実行による誤り発見から明示的なテスト実行による誤り発見までの差を示す．テストの失敗は，まず自動的なテスト実行により記録され，その後，ユーザによる明示的な実行によりユーザに失敗が示される．自動的なテスト実行によるテスト失敗の記録からユーザによる明示的なテスト実行によるテスト失敗までの編集回数を，自動的なテスト実行による早期発見の効果として測定する．自動的なテスト実行による同一のテスト失敗が n 回繰り返された後でユーザによる明示的なテスト実行によりそのテスト失敗がユーザに示された場合，ユーザは誤りの存在を知らないまま $n-1$ 回の編集作業を行なったことになる．これは，自動的なテスト実行はユーザによる明示的なテスト実行より $n-1$ 回の編集作業分，早期に誤りを発見できることを意味する．

テスト失敗には，仕様の誤りによるものと，テストの誤りによるものがある．表 1 では，誤りの所在として示した．自動的なテスト実行によって発見された仕様の誤りが 5 つのうち，3 つの仕様の誤りについて，ユーザーによるテスト実行による発見が遅れた．特に誤り 12 については，自動的なテストの実行から 9 回分，ユーザーによるテスト実行による発見が遅れた．この時ユーザーは編集対象のモジュールのテストを毎回実行していたが，そのモジュールを利用している別のモジュールのユニットテストでエラーが発生していた．自動的に全テストモジュールを実行していれば，実際よりも修正 9 回分だけ早期にそのエラーを認識することができた可能性がある．

誤り 9 は，ユーザーの操作ミスによって，意図しないモジュールに対して意図しない編集が保存されてしまったために発生した．そのためユーザーがそのモジュールに対するテスト実行の必要性を認識せず，一通りの作業が終了した後に念のために実行した全テストモジュール

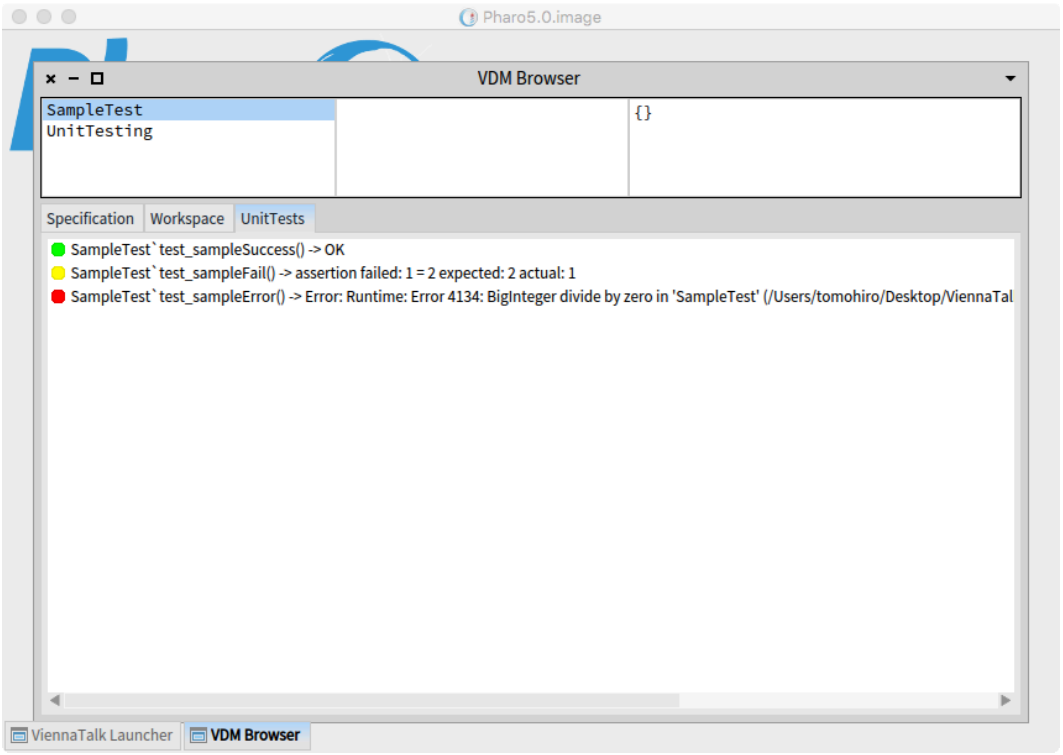

図 5. VDMBrowser でのテスト結果一覧の表示例

表 1. 自動的なテスト実行と明示的なテスト実行による誤り発見時期の差

誤り	失敗またはエラー	誤りの所在	発見時期の差	誤りの概要
誤り 1	失敗	テストの誤り	0	テストデータの誤り
誤り 2	エラー	テストの誤り	0	テストデータの誤り
誤り 3	失敗	仕様の誤り	0	操作中の処理の欠落
誤り 4	エラー	テストの誤り	0	テスト操作中での処理の欠落
誤り 5	失敗	テストの誤り	0	表明の参照先の誤り
誤り 6	エラー	テストの誤り	0	テストデータの誤り
誤り 7	エラー	仕様の誤り	0	場合分けの不足
誤り 8	エラー	仕様の誤り	0	場合分けの不足
誤り 9	エラー	仕様の誤り	4	場合分けの不足
誤り 10	エラー	テストの誤り	1	import 宣言の誤り
誤り 11	エラー	テストの誤り	0	テスト操作中での処理の欠落
誤り 12	エラー	仕様の誤り	9	場合分けの不足
誤り 13	エラー	テストの誤り	0	テスト操作中での処理の欠落

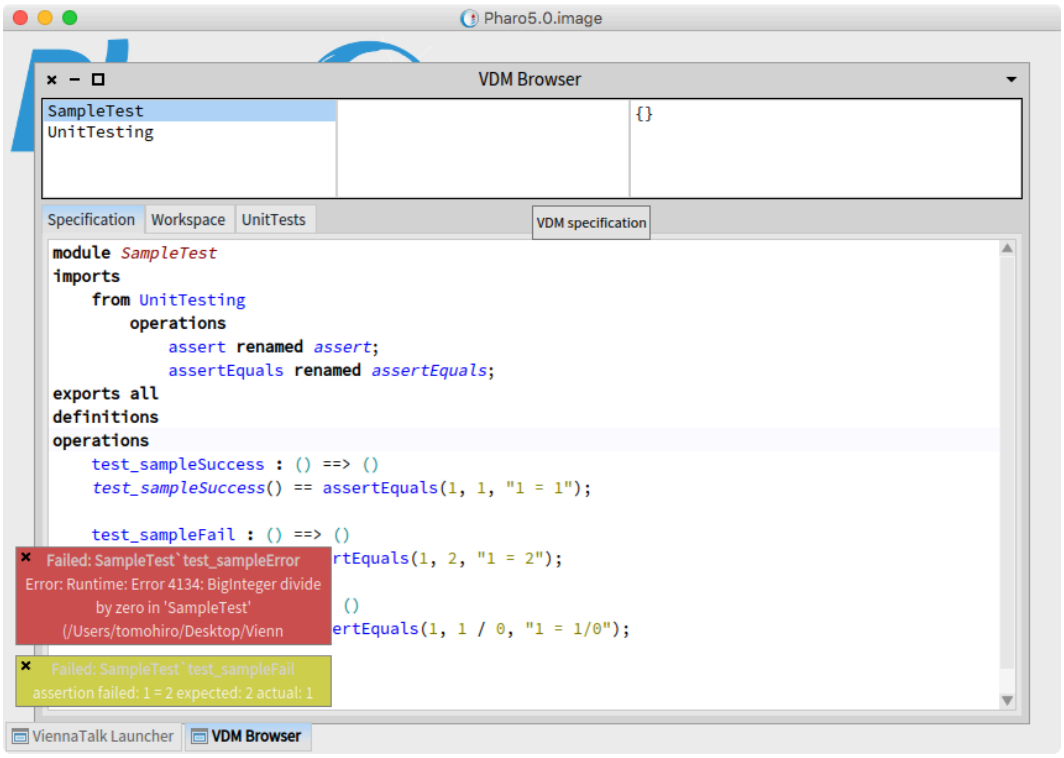

図 6. ViennaTalk におけるテスト失敗通知の表示例

実行によって誤りを発見した．誤り 10 は，テスト操作を 2 つ連続して記述する作業で，1 つ目のテスト操作を保存し，それに誤りがあったところを，そのまま 2 つ目のテスト操作の記述を開始したことにより，誤りの発見が 1 回分遅くなった．しかし，2 つ目のテスト操作を保存した後でそのテストモジュールを実行したため，1 回分の遅れのみで比較的早期に発見することができた．

以上より，ViennaUnit の自動的な実行は，仕様の誤りの早期発見に有効であると考えられる．一方，テストの誤りに関しては，必ずしも自動的な実行をしなくても，開発環境からユニットテストを明示的に実行することで早期発見された．これはテストがモジュール間の独立性が高いことが原因と考えられる．仕様を編集した場合には，編集の影響がモジュールをまたがって及ぶため，編集したモジュールのみをテストしても誤りを見逃すことがある．テストは仕様と比較してテストモジュール間の独立性が高いために，編集したテストのみを即座に実行することで，自動的なテスト実行にほとんど遅れることなく発見されたと考えられる．

6. 議論

ViennaTalk は探索的仕様記述を支援するための環境であり，本稿で紹介したテストフレームワークも探索的仕様記述での試行錯誤の中でより早期に誤りを発見し修正することを目的としている．テストフレームワーク ViennaUnit は xUnit から派生したテストフレームワークの簡易な実現であり，共通の機能が多い．形式仕様に対するユニットテストとプログラムコードに対するユニットテストには技術的には多くが共通しているが，目的は大きく異なっている．テスト駆動開発では，テストコードはプログラムコードの仕様として扱われる．一方で，形式仕様に対するユニットテストはテスト対象の仕様ではない．仕様がテスト対象であり，テストは仕様が定義する機能に関するシナリオを通して仕様記述者やドメイン専門家の理解と機能モデルの間の整合性を確認するための記述である．ViennaUnit が自動的なテスト実行により早期に誤りを発見することで，仕様記述者は記述内容と理解の不整合を認識することができる．

VDMBrowser によるテストの自動実行は，ユーザーが

仕様記述に集中することを助ける点でも効果的である．テスト実行の処理時間はテスト操作の定義に依存するが，第 5 節で評価した仕様およびテストでは，52 個のテスト操作の実行に約 30 秒かかった．仕様のモジュールまたはテストモジュールを編集するたびにメニューからテスト実行を選択して 30 秒待つことは，ユーザーである仕様記述者の集中を損ねる．テストの自動実行および邪魔にならないテスト失敗通知により，編集するたびに 30 秒待つことなく，仕様記述を継続することができる．

VDMBrowser でのテスト実行の課題としては，後工程との連携が挙げられる．仕様に対するテストの目標は，仕様テストの失敗やエラーが発生しなくなることではない．仕様として記述した内容が仕様記述者やドメイン専門家の理解と齟齬がないことを確認し，適切な仕様を記述することで後工程での精緻化により厳密な仕様を作成し，品質の高いソフトウエアの実装を出荷することが目標である．そのためには，実装に対するテストやデバッグにおいて，仕様に対するユニットテストのテストモジュールの定義やテスト結果が後工程で有効に利用可能であることが望ましい．

7. まとめ

探索的仕様記述を支援するための仕様記述環境 ViennaTalk でのテストフレームワーク ViennaUnit を説明し，評価した．ViennaTalk は探索的仕様記述を前提とした記述環境であるが，最終的な目的は実装されたソフトウェアの生産性や品質の向上である．そのためには，探索的仕様記述の後工程である精緻化や実装での開発手法やそのためのツールとの連携が求められる．今後は，精緻化や実装との連携を中心に，ViennaTalk に求められる機能やツール連携を分析し，開発を継続する．

参考文献

[1] J. Woodcock, P. G. Larsen, J. Bicarregui and J.. Fitzgerald, “Formal Methods: Practice and Experience”, *ACM Computing Surveys*, Vol. 41, No. 4, pp. 1–36, 2009.

[2] N. Ubayashi, S. Nakajima and M. Hirayama “Context-dependent product line engineering with lightweight formal approaches”, *Science of Computer Programming*, Vol. 78, Issue 12, pp. 2331–2346, 2012.

[3] J. Fitzgerald and P. G. Larsen “Modelling Systems – Practical Tools and Techniques in Software Development”, Cambridge University Press, 1998.

[4] K. Lausdahl, P. G. Larsen and N. Battle “A Deterministic Interpreter Simulating A Distributed real time system using VDM”, *Proceedings of the 13th international conference on Formal methods and software engineering*, pp. 179–194, 2011.

[5] T. Kurita and Y. Nakatsugawa “The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip”, *Intl. Journal of Software and Informatics*, Vol. 3, No. 2-3, pp. 343–355, 2009.

[6] 小田朋宏, 荒木啓二郎 “形式仕様工程の初期段階に着目した統合仕様記述環境 ViennaTalk”, コンピュータソフトウェア, 34 巻, 4 号, ppp. 4_129-4_143, 日本ソフトウェア科学会, 2017.

[7] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, Damien and M. Denker “Pharo by Example”, Square Bracket Associates, 2009.

[8] J. Fitzgerald, P. G. Larsen and S. Sahara “VDMTools: advances in support for formal modeling in VDM”, *ACM Sigplan Notices*, Vol. 43, No. 2, pp. 3–11, 2008.

[9] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl and M. Verhoef “The Overture Initiative – Integrating Tools for VDM”, *SIGSOFT Softw. Eng. Notes*, Vol. 32, No. 1, pp. 1–6, 2010.

[10] P. Hamill “Unit Test Frameworks”, O’Reilly, 2004.

[11] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat and M. Verhoef “Validated Designs For Object-oriented Systems”, Springer, 2005.

[12] P. G. Larsen, K. Lausdahl, P. W. V. Tran-Jørgensen, A. Ribeiro, S. Wolff and N. Battle “Overture VDM-10 Tool Support: User Guide”, The Overture Initiative, TR-2010-02, 2010.